

A Traversable Fixed Size Small Object Allocator in C++

Christian Schüßler^{a,b}, Roland Gruber^b

{roland.gruber and christian.schuessler}@iis.fraunhofer.de

^aTechnische Hochschule Nürnberg

^bDepartment production monitoring, Fraunhofer EZRT

Abstract

At the allocation and deallocation of small objects with fixed size, the standard allocator of the runtime system has commonly a worse time performance compared to allocators adapted for a special application field.

We propose a memory allocator, originally developed for mesh primitives but also usable for any other small equally sized objects.

For a large amount of objects it leads to better results than allocating data with the C++*new* instruction and behaves nowhere worse. The proposed synchronization approach for this allocator behaves lock-free in practical scenarios without using machine instructions, such as compare-and-swap.

A traversal structure is integrated requiring less memory than using containers such as STL-vectors or lists, but with comparable time performance.

Keywords: small object allocation, traversable memory allocator, non-blocking memory allocator

1. Introduction

At the allocation and deallocation of huge amounts of small objects, a specialized memory allocator leads to better performance than using the standard allocator of the runtime system. The purpose of this algorithm is to allocate and deallocate mesh primitives. This means vertices, edges, and faces. A mesh often consists of millions of them. However, this allocator can be applied to any application having the same requirements.

We give a brief introduction to the data structures of the implemented memory allocator in the first section. Afterwards we describe how to preserve the consistency of the traversing data structure in the allocator after a deallocation. For multithreaded applications a synchronization approach is proposed. The result section shows the time performance and memory consumption of the implemented allocator.

2. Concepts and Data Structures

A common approach for custom memory allocation is to pre-allocate a large memory block and split it into small pieces, which are then assigned to the user. This is often called region-based-memory management [1, 2, 3, 4]. We call a memory block a *bin*, the small pieces corresponding to the individual requested objects are called *chunks*.

Allocators that use this approach are considered to be faster than using the default allocator, but it also leads to higher memory consumption [5]. This allocator is designed for a typical chunk size of 32 bytes (for example, four double values). Due to the result section, we suggest a bin size of about 32,000 elements. Therefore a bin consumes about 1 MB including meta data. This amount of memory is negligible in our application field.

Listing 2 shows the internal data structure of a single chunk. The chunk struct consists of a content element (*ContentElement* in line 13) and a status field (*status* in line 22). The most significant bit of the status field marks whether the assigned chunk is free or assigned. The remaining bits are used to save the position of the next free chunk, this data structure is known as *free-list* [6, p. 36f]. It is used to get a free chunk for allocation without additional searching for a free element in the memory allocator.

If the chunk is free, its memory is interpreted as doubly linked list (line 16), storing the next and previous assigned chunk. Otherwise the chunk stores the actual data assigned to the user (line 15). The idea to use free chunks to store a doubly linked list is similar to Doug Lea’s memory allocator [7] and enables a fast traversal over all assigned chunks.

The smallest size of a chunk is 20 bytes on a 64-bit system. Two list pointers (16 bytes) and the status field (4 bytes). An example with a content size of 24 bytes is shown in Figure 1.

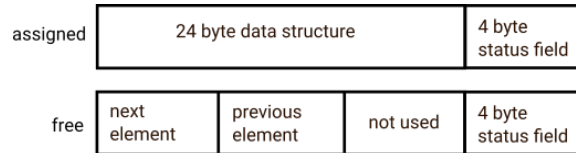


Figure 1: The data configuration of an assigned chunk (top). The configuration for a free chunk, with two required list pointers (bottom). In both configurations a 4-byte status field is used.

```

1  class Chunk;
2  struct LinkedList
3  {
4      Chunk*  next;
5      Chunk*  previous;
6  };
7
8  struct InternalContentType
9  {
10     uint8  content[sizeof(ContentType)];
11 };
12
13 union ContentElement
14 {
15     InternalContentType content;
16     LinkedList  linkedList;
17 };
18
19 struct Chunk
20 {
21     ContentElement element;
22     uint32 status;
23 };

```

Listing 2: internal data structure of *Chunk*

A bin holding several chunks is shown in Figure 3.

Only the list entries of free chunks located at the boundary of assigned chunks need to be correct. These free chunks hold the address of the next assigned chunk and all other free chunks between can be skipped. This approach makes a fast traversal possible because not every single free chunk has to be checked. Figure 5 shows an example configuration of the memory allocator. All bins are connected through a doubly linked list to enable traversal in both directions.

Pseudo elements are used to make a traversal from the beginning to the end of a bin possible. Also they allow us to ignore special cases, such as if there only exists free chunks in a bin. There is always a chunk marked as assigned at the beginning and at the end of each bin. The chunk right after or before is marked as free. These four pseudo elements are only used by the allocator and are neither visible nor allocatable by the user. Figure 5 shows several bins, illustrating this configuration.

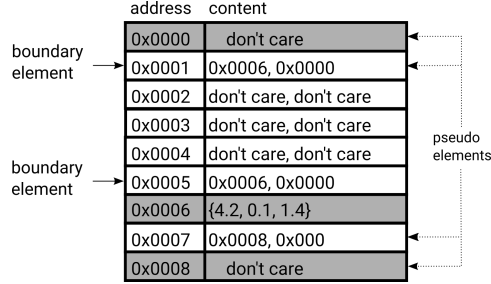


Figure 3: In this example bin, an array of float values is stored as user content. The white entries mark free chunks and the gray assigned ones. List entries are only required by free chunks at the boundary to an assigned chunk. The address column is not part of the data structure but is used to illustrate the traversal pointers. Don't care fields mean that the actual content of the chunk is not relevant.

3. Traversing Correction Algorithm

The previous section described the data structure that enables a fast traversal over all assigned chunks. However, after each allocation or deallocation the pointers in the free chunks might be invalid. For example, if a free chunks gets assigned, the linked list of two free chunks, located before and after, must be corrected to point to this new assigned chunk.

In this section we show how the structure can be corrected at an allocation or deallocation call, with time complexity $O(1)$, because we avoid any searching in the data structure. To make further explanations more clear, some definitions are needed.

A pointer in the linked-list of a free chunk is *valid* if it is the *next* member in the linked-list and points to the next assigned chunk or if it is the *previous* member in the linked-list and points to the previous assigned chunk.

The structure is *valid* if

1. The *previous* pointer in the linked-list of every free chunk directly before an assigned chunk is *valid*

2. The *next* member in the linked-list of every free chunk directly after an assigned chunk is *valid*

If these conditions hold, traversing over all assigned elements is possible. The structure has to be *valid* again, after each allocation or deallocation call. To correct the data structure in $O(1)$ time, some further conditions must hold.

1. In an initialized bin, both pointers in the linked-list in every chunk are *valid*.
2. The free-list is initialized in such a way that elements are assigned in consecutive order.
3. If an element is deallocated, both members in the linked-list in the freed chunk are set valid and the chunk is pushed to the top of the free-list.

Following these conditions, the correction at the allocation and deallocation call can be implemented with only a few if-clauses and write operations. Figure 5 shows a valid memory allocator configuration, following all the previous stated conditions.

The allocation algorithm is shown in listing 4. In the first line, the position of the new element is taken from the free-list. The position of the chunk, which is taken at the next allocation call, is set in line two. The next two if-clauses are required because there might be free-chunks which are pointing to invalid chunks. This is illustrated in Figure 6 for the correction of the previous pointer.

```

1  Chunk* newElement = _baseAddress + _nextFreeElement;
2  _nextFreeElement = newElement->status & (~FREE_FLAG);
3
4  if((newElement - 1)->status & FREE_FLAG)
5      (newElement->element.linkedList.previous + 1)->element.linkedList
        .next = newElement;
6
7  if((newElement + 1)->status & FREE_FLAG)
8      (newElement->element.linkedList.next - 1)->element.linkedList.
        previous = newElement;

```

Listing 4: Correction of the structure at allocation

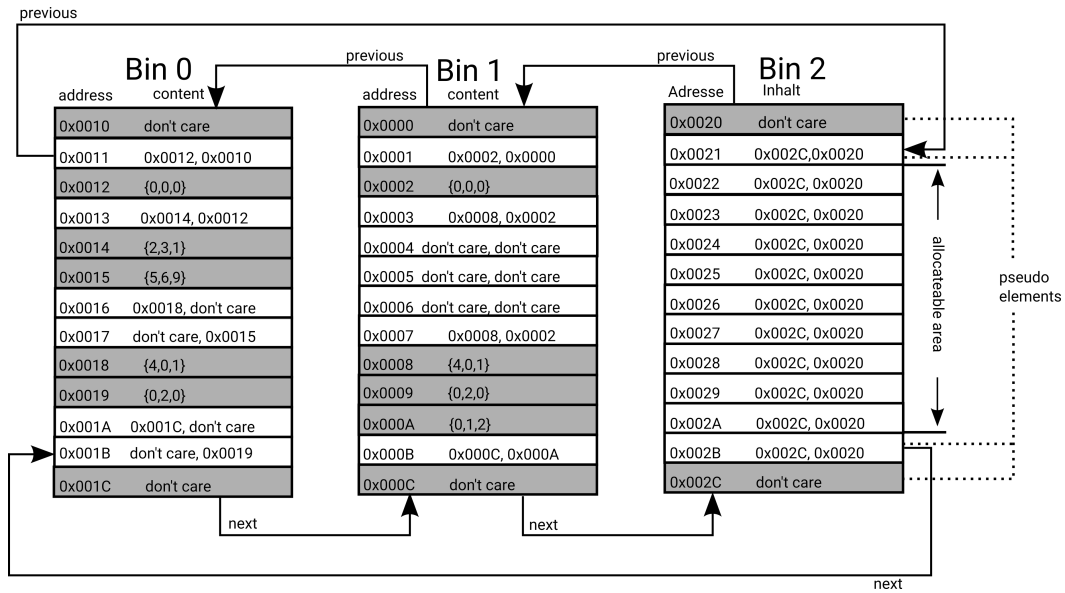


Figure 5: Example of the data containers used by the small object allocator. The bins holding the chunks are connected through a doubly linked-list. The right bin represents a new initialized empty bin.

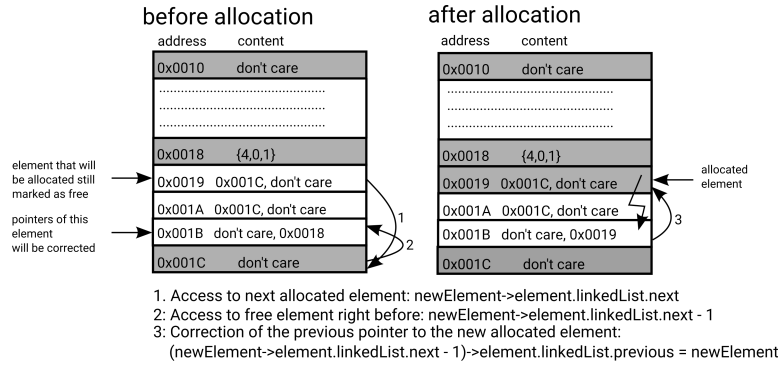


Figure 6: Correction of the previous pointer to the new assigned chunk

For deallocation, the structure has to be corrected with two if-clauses exactly as in the allocation call, to leave the structure valid. The deallocated chunk has to be marked as free and pushed to the top of the free-list, so it can be assigned at the next allocation call, see line 1-2 in Listing 7. The next two conditions in lines 4-12 are necessary to make both pointers in the now free chunk valid. This is because the linked-list of a free chunk assigned in an allocation call has to be valid, see lines 4-8 in Listing 4.

```

1 chunk->status = _firstFreeElement | FREE_FLAG;
2 _firstFreeElement = static_cast<uint32>(chunk - _baseAddress);
3
4 if ((chunk + 1)->status & FREE_FLAG)
5     chunk->element.linkedList.nextElement = (chunk + 1)->element.
6     linkedList.nextElement;
7 else
8     chunk->element.linkedList.nextElement = (chunk + 1);
9
10 if ((chunk - 1)->status & FREE_FLAG)
11     chunk->element.linkedList.previousElement = (chunk - 1)->element.
12     linkedList.previousElement;
13 else
14     chunk->element.linkedList.previousElement = (chunk - 1);

```

Listing 7: Correction of the data structure at deallocation.

Correcting the structure in that way, leaves it valid after each allocation or deallocation. This is shown in the following.

After initialization of a bin, it is valid and also every chunk, see Figure 8. After the first object has been assigned, the previous and the next pointer of

the chunk before are both valid, as well as the next pointer of the following chunk. This is sufficient for a valid data structure. A deallocation of this element would lead to the same state in the data structure as when it was initialized, because after a deallocation call, both pointers are made valid. At the allocation of all elements after the first one (lowest memory address), only the free chunk located after this element has to be valid, because the previous chunk is always assigned if no deallocation appeared. However, the user can deallocate chunks in arbitrary order. In this case the last deallocated chunk will be set valid and its pointer will be pushed on top of the free-list. If this deallocated object is assigned again, the valid linked-list can be used to correct the structure as already shown in listing 4. The problem is that after the second deallocation, the pointers in the linked list of the deallocated chunk before are getting invalid. So it might be assumed, that this element can never be used to correct the traversing data structure. This problem can be avoided by allocation in reverse order to deallocation. Therefore an allocation leads to exactly that state in which this chunk was valid. So both pointers in its linked list point to the nearest assigned chunk, as explained in the beginning of this section. An example of arbitrary allocation and deallocation calls is shown in Figure 8.

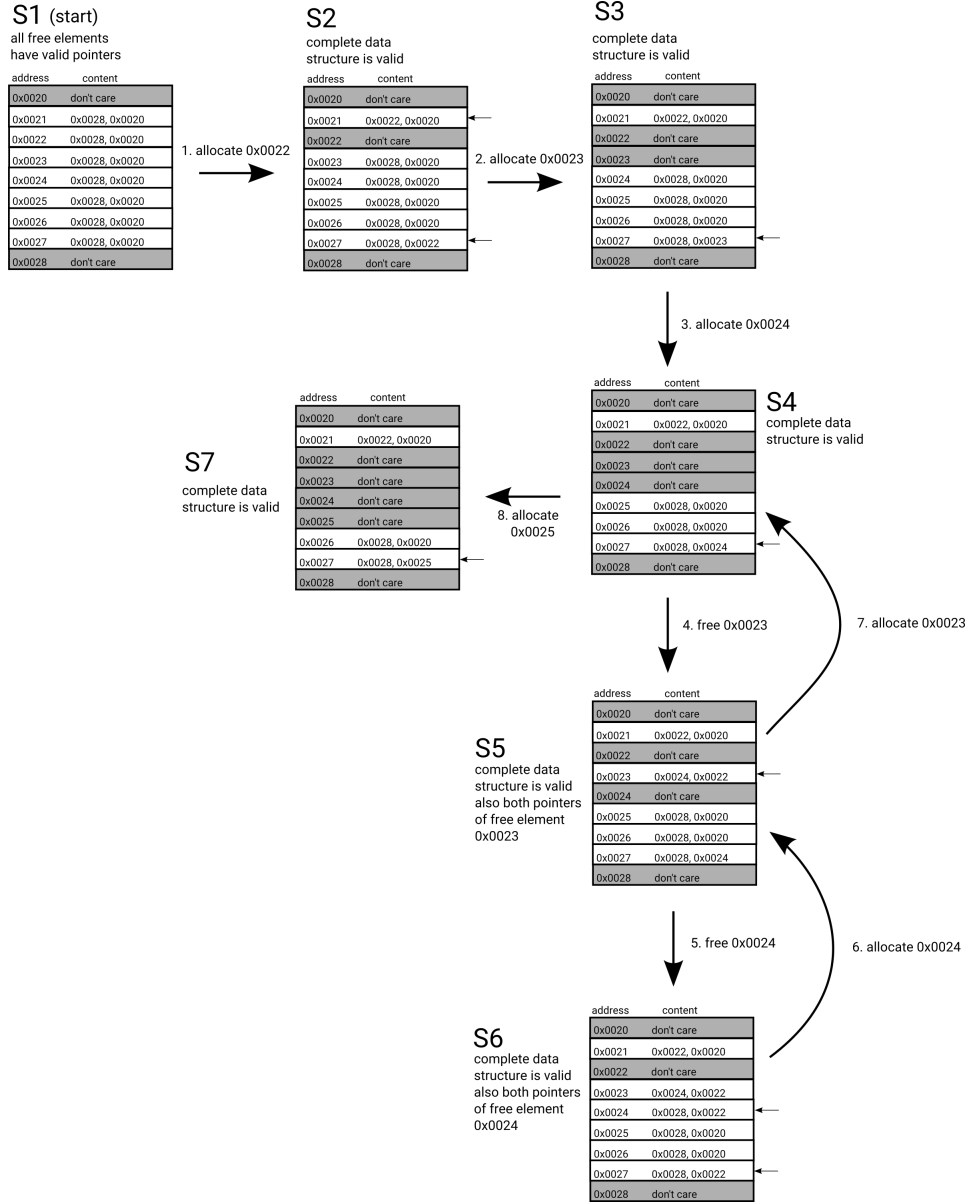


Figure 8: Some allocation and deallocation calls changing the state of the data structure. The arrows mark a deallocation or allocation call. The Symbol **SX**, where **X** is a number, marks the current state with description. The small arrows pointing to chunks in a bin mark where the content of a chunk has changed.

4. Thread-Safe-Allocation using readers-writers-locks

In this section, we propose a synchronization approach that behaves lock-free in most practical applications.

The idea is to synchronize the doubly linked-list of bins by using a readers-writers-lock and lock each bin separately using a mutex. The current thread checks for each bin if it is full or locked and gets to the next bin if necessary. Also every thread remembers the last bin where it has assigned a chunk. Therefore the most common case is that every thread allocates memory in a different bin. In this case, allocation works without blocking. Only if a new bin has to be inserted, the complete list must be locked. The probability of this event drops with increasing bin size. Assigning each thread to a local bin is commonly known as *thread local allocation* [8]. The complete allocation process is summarized in Figure 9. Our small object allocator has a complexity of $O(n)$ at allocation, due to traversing other bins if necessary. This is negligible if the ration of number of allocated chunks to bin size is large enough.

Memory can only be completely deallocated if all chunks in a bin are marked as free.

In [9] a similar concept for non fixed size elements, using lock-free singly-linked-lists, is proposed. We avoid the usage of completely lock-free data structures, because of the need of machine dependent compare-and-swap instructions and the dealing with the ABA-problem [10].

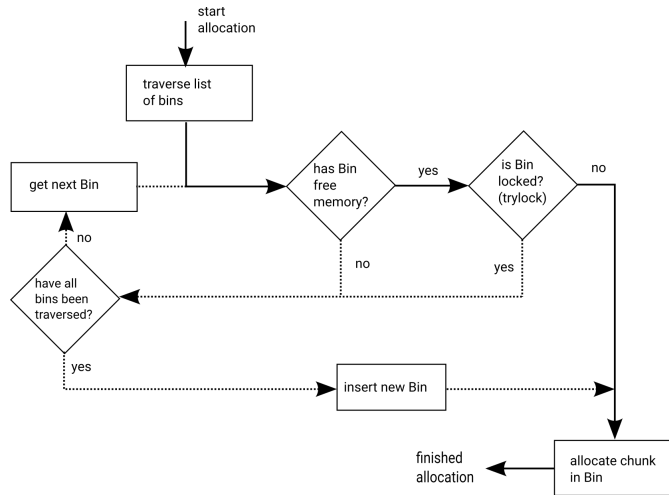


Figure 9: Complete allocation scheme for the memory allocator. The solid line marks the most likely path.

5. Multi-Thread Iteration

The bins in the memory allocator can be distributed into several bin lists, which can be iterated independently. Each bin list can be assigned to a single thread, so that the iteration speed should be increased by the number of threads. Each begin-iterator of a bin list stores the first assigned chunk in the first bin. Each end-iterator stores the last chunk in the last bin. Figure 10 gives an example for distributing the bins into two lists.

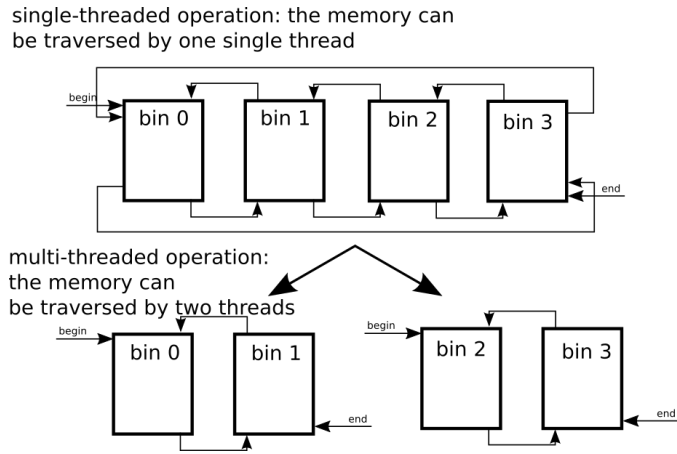


Figure 10: The memory allocator is divided into several bin lists. Each pair of begin and end-iterator is assigned to a single bin list. This division allows for all elements in the memory allocator to be iterated independently.

6. Results

Other small object allocation approaches can be found in [11, 12, 13, 14, 15]. The boost *memory pool* uses the same concepts as mentioned in [16] and guarantees a working implementation in a well tested library. Therefore it was decided to compare the small object allocator with this implementation. The data type used for the small object is an array of three unsigned integer(64-bit) values. This data type has a typical size of 24 bytes for which the allocator was designed.

The test system specifications are stated in Table 1.

processor	Intel Core i7-4770 CPU @ 3.40 GHz
main memory	4 cores, hyperthreading switched off
operation system	16 GB
	Windows 7 Enterprise

Table 1: Hardware specifications of the test platform

The first benchmark was to allocate 200 million instances of this integer-array type. Each allocator was called in a simple for-loop for each object instance. The small object allocator was tested with different numbers of chunks as bin size. The result is shown in Figure 11.

In all following tests the bin size was set to 64,000 chunks. At this bin size the allocator reached its optimum and is almost as fast as the boost implementation. A fairer way is to compare the allocator with the additional use of STL-containers by the boost memory pool, to make an iteration also for these allocators possible. In this case the small object allocator is clearly faster.

The next benchmark measures the performance with concurrent accesses through multiple threads, this is shown in Figure 12. In this case an OpenMP for-loop with different numbers of threads was used. The synchronization technique proposed in section 4 is much faster than normal mutex locking and behaves like a non-blocking synchronization.

Table 2 shows the memory consumption of the complete process. The implemented allocator has the best results. It does not need an additional traversal structure. However, the boost memory pool and the default C++ new operator do not differ a lot.

The results of the benchmark for the implemented iterators, using the integrated traversal structure are shown in Table 3. In every case 200 million elements were iterated. Having no fragmentation (no gaps between the assigned chunks) the implemented allocator leads to a performance that is

between the STL-vector and the STL-list. Additionally, memory gaps were generated in deallocating an element randomly, with the likelihood of 10 and 50%. Deallocation at random positions is the worst case scenario for the implemented traversal, because unallocated memory has to be skipped and therefore some extra checks have to be done. A more common case is to free a bunch of elements locally near to each other in memory. This behavior should lead to a better time performance.

We also benchmarked the parallel iteration with 1 to 8 threads, as shown in Figure 13. Each iterator executes the code in Listing 15, which simulates a time-consuming algorithm. The image shows that the parallel iteration scales well up to 4 threads, which corresponds with the number of cores in the test environment. Cache-misses were examined by measuring all requests that missed the L2 cache [17], see Figure 14. There is no hint of false-sharing, because the number of cache-misses does not differ drastically between the different measurements.

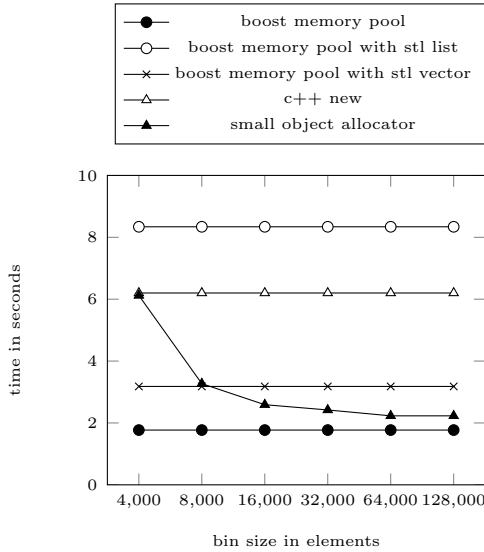


Figure 11: Allocation using the implemented small object allocator with different bin sizes compared to boost memory pool and default new operator of the C++ language. All allocators beside ours were measured once but were plotted for each bin size. Because they have no bin size as parameter.

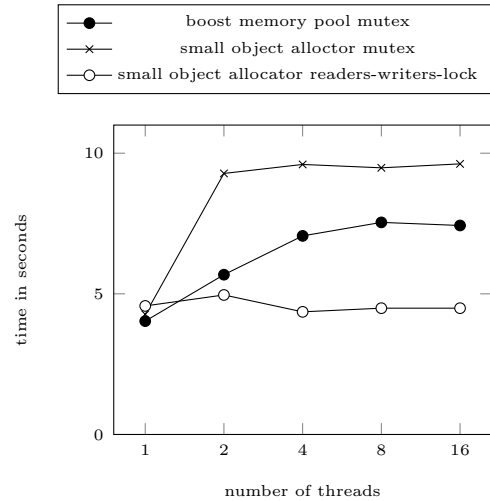


Figure 12: Allocation using the boost memory pool and the small object allocator with different synchronization approaches.

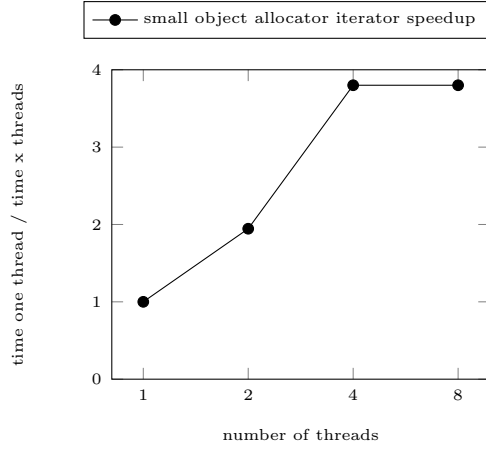


Figure 13: Performance benefit by using parallel iteration at a memory bin size of 64,000 elements.

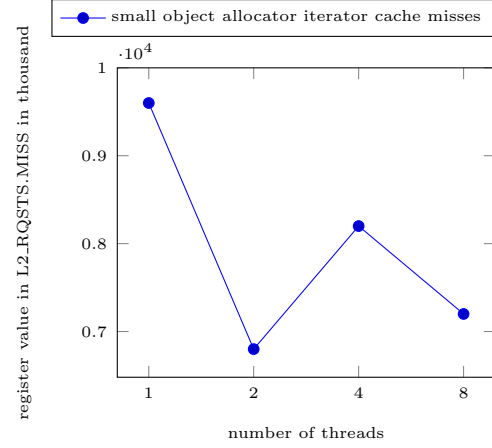


Figure 14: Cache-misses for different numbers of threads.

Allocator	Memory consumption in MB
boost memory pool	6,308
boost memory pool with vector	7,876
boost memory pool with list	12,584
C++ new	6,279
small object allocator	6,276

Table 2: Memory consumption comparing different allocator types.

Container	Iteration time in seconds
STL-vector	0.64
STL-list	1.15
small object allocator	0.64
small object allocator (10% gaps)	0.79
small object allocator (50% gaps)	1.84

Table 3: Iteration time using different container classes.

```

1  const std::uint32_t RANDOM_VALUES = 20;
2
3  std::uint32_t * randomValues = new std::uint32_t[RANDOM_VALUES];
4  srand(clock());
5
6  for(std::uint32_t i = 0; i < RANDOM_VALUES; i++)
7      randomValues[i] = rand() % 10;
8
9  for(std::uint32_t i = 0; i < RANDOM_VALUES; i++)
10 {
11     it->x += randomValues[i];
12     it->y += randomValues[i] * 2;
13     it->z += randomValues[i] * 4;
14 }
15 delete [] randomValues;

```

Listing 15: A small benchark applied for every element in the memory allocator.

7. Summary

We presented an allocator for fixed sized small objects that allows fast traversal over its elements.

The performance of the implemented allocator is comparable with the widely used boost memory pool at single-threaded applications. In applications where traversal is required and low memory consumption is favorable, the implemented allocator is superior. Additionally, the proposed synchronization approach leads to a non-blocking behavior in practical applications. The allocator is very well behaved for parallel iteration.

8. Acknowledgements

I am grateful to Jörg Arndt for several reviews and for supervising my bachelor thesis [18] that led to this publication. I also want to thank Edith Parzefall for proofreading.

References

- [1] D. Gay, A. Aiken, Memory management with explicit regions, volume 33, ACM, 1998.
- [2] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, J. Cheney, Region-based memory management in cyclone, ACM Sigplan Notices 37 (2002) 282–293.
- [3] M. Tofte, J.-P. Talpin, Region-based memory management, Information and computation 132 (1997) 109–176.
- [4] M. Tofte, L. Birkedal, M. Elsmann, N. Hallenberg, A retrospective on region-based memory management, Higher-Order and Symbolic Computation 17 (2004) 245–265.
- [5] E. D. B. B. G. Zorn, K. S. McKinley, Reconsidering custom memory allocation (2002).
- [6] P. R. Wilson, M. S. Johnstone, M. Neely, D. Boles, Dynamic storage allocation: A survey and critical review, in: Memory Management, Springer, 1995, pp. 1–116.
- [7] D. Lea, W. Gloger, A memory allocator, 1996.

- [8] E. D. Berger, K. S. McKinley, R. D. Blumofe, P. R. Wilson, Hoard: A scalable memory allocator for multithreaded applications, *ACM Sigplan Notices* 35 (2000) 117–128.
- [9] M. Aigner, C. M. Kirsch, M. Lippautz, A. Sokolova, Fast, multicore-scalable, low-fragmentation memory allocation through large virtual memory and global data structures, *ACM SIGPLAN Notices* 50 (2015) 451–469.
- [10] M. Helrihy, N. Shavit, *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2012.
- [11] D. Bulka, D. Mayhew, *Efficient C++: performance programming techniques*, Addison-Wesley Professional, 2000.
- [12] J. Coplien, Advanced c++ programming styles and idioms, in: *Technology of Object-Oriented Languages and Systems, 1997. TOOLS 25, Proceedings*, IEEE, pp. 352–352.
- [13] A. Alexandrescu, *Modern C++ Design*, Addison-Wesley, 2007.
- [14] B. Stroustrup, *The C++ programming language*, Pearson Education India, 1986.
- [15] J. Labrosse, *Microc/os-ii*, R & D Books 9 (1998).
- [16] S. Cleary, P. A. Bristow, *C++ boost memory pool documentation appendix f - other implementations*, 2015.
- [17] I. Corporation, *Intel 64 and IA-32 Architectures Software Developer’s Manual*, volume 3b: system programming guide, part 2 edition, 2013.
- [18] C. Schüßler, *Entwurf und Implementierung einer effizienten Speicherverwaltung für Mesh-Primitive*, Bachelorthesis, Technische Hochschule Nürnberg Georg Simon Ohm, 2014.